Unbounded Safety Verification for Hardware Using Software Analyzers

Rajdeep Mukherjee
University of Oxford

Peter Schrammel
University of Oxford

Daniel Kroening
University of Oxford

Tom Melham
University of Oxford

***Abstract*—Demand for scalable hardware verification is ever-increasing. We propose an unbounded safety verification framework for hardware, at the heart of which is a software verifier. To this end, we synthesize Verilog at register transfer level into a *software-netlist*, represented as a word-level ANSI-C program. The proposed tool flow allows us to leverage the precision and scalability of state-of-the-art software verification techniques. In particular, we evaluate unbounded proof techniques, such as predicate abstraction, *k*-induction, interpolation, and IC3/PDR; and we compare the performance of verification tools from the hardware and software domains that use these techniques. To the best of our knowledge, this is the first attempt to perform unbounded verification of hardware using software analyzers.**

I. Introduction & Background

With the ever-increasing complexity of hardware and SoC-based designs for mobile platforms, demand for scalable formal verification tools in the hardware industry is always growing. The scalability of hardware model checking tools depends on three key factors: the *design representation*, the *verification engine*, and the *proof engine*. This paper experimentally evaluates the influence of the first two factors: the design representation, and the verification engine. Figure 1 shows the phases that a typical hardware model checking tool passes through.

Design representation: Given a hardware design in Verilog RTL, formal verification tools use different internal representations for the design, at differing levels of design granularity: *bit level*, *word level*, *term level* or *software level*. Figure 1 lists some of the design representations commonly used. Most formal verification tools for hardware [4], [6], [7] synthesize the input design into a bit-level netlist, typically represented as *AIG*, and stored in formats such as *BLIF*, *EDIF*, *PLA* or *BAF*. This approach misses the opportunity to exploit the word-level structure of the input RTL design. Tools based on word-level representations may use *BTOR* or another intermediate word-level format, which enables the use of word-level decision procedures, such as Satisfiability Modulo Theory (SMT) solvers, in the back-end of these tools.

The Verification Engine: Over the past few years, we have seen that verification tools participating in Hardware Model Checking Competition (HWMCC) employ a variety of verification engines to speed up proofs or to detect deep bugs. After McMillan's notable work on interpolation-based model checking [19], the work of Bradley [4] on incremental inductive invariant generation (IC3) proved to be a paradigm shift in scaling bit-level hardware verification. The success of IC3 is its ability to perform unbounded verification as well as quickly detect deep bugs based on relative induction strengthening. IC3 or Property Directed Reachability (PDR), *k*-induction, and interpolation-based approaches have all made their way from the hardware to the software domain; but predicate abstraction and abstract interpretation have remained primarily used for software verification.

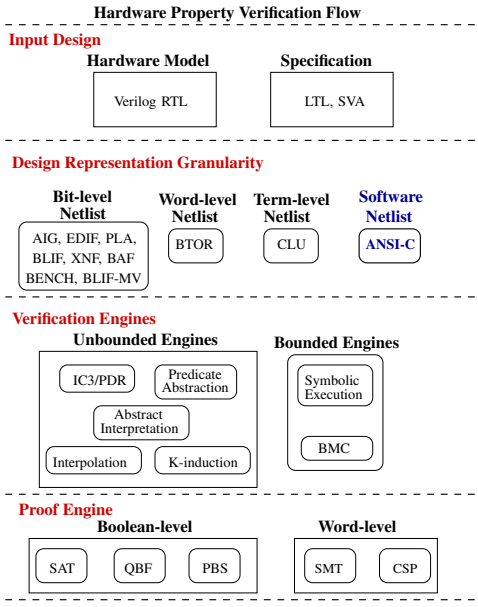

Fig. 1. Conventional flow for hardware property verification

The use of high-level structures as a design description for model checking is a holy grail of hardware verification [8], [9], [14], [15], [20], [22]. The efficiency and scalability of the verification engine depends on the granularity of design descriptions as well as the decision procedures used. This is exemplified by the scalability of word-level hardware verification [16], [22], tools such as word-level STE [8], BMC [2], [20], predicate abstraction [14] and interpolation [17] over the corresponding bit-level implementations. Recent work described in [10], [24] generalizes PDR to richer logics such as QF_BV and the software domain—and demonstrates better scalability than bit-level PDR.

In this paper, we present a novel unbounded safety verification tool flow for hardware. Unlike traditional approaches that synthesize the design into a bit-level netlist [4], [6], [7] or generate an abstract model of the design (say C/C++

Supported by EPSRC EP/J012564/1, ERC project 280053 and the Semiconductor Research Corporation (SRC) under task 2269.001.

ISA or micro-architectural models derived from RTL [9]), we propose an orthogonal approach for hardware verification using *software verifiers*. To this end, we automatically synthesize the hardware models, articulated in Verilog at register transfer level, into a *software-netlist*, represented as word-level ANSI-C program. The generated model is *bit precise* and *cycle accurate*, but expressed as a software program. This opens the door to exploiting the full range of unbounded software verification techniques in a common framework.

II. Contributions

In this paper, we report the following contributions:

1) *Unbounded Hardware Verification using Software Analyzers:* We present an unbounded safety verification tool flow for hardware IPs using software analyzers. To this end, we perform automatic synthesis of Verilog RTL into a software-netlist.
2) *Unified and Integrated Framework:* One advantage of the proposed tool flow is that it provides a unified framework to evaluate various unbounded verification techniques at the bit level, word level, and software level. This enables us to compare model checking tools from the Hardware Model Checking Competition (HWMCC)[1] with tools from the Software Verification Competition (SV-COMP)[2].
3) *Benchmarking:* Although SoC designs are increasingly written at a higher level [15], [18], there is still a significant body of existing design IP blocks that are written in VHDL or Verilog. The proposed tool flow allows rapid generation of software-netlist models from hardware IP derived from real-world hardware benchmark suites.

III. Unbounded Hardware Verification

A. Generating Software-Netlist from Verilog

Given a hardware design in Verilog RTL, our tool *v2c* [20] synthesizes the design into a software-netlist model in C. Each state transition in hardware can be viewed as set of register updates according to the next-state function and assignment of non-deterministic values to the external inputs. In the software-netlist, the state transition is also achieved by updating the sequential or state-holding elements and explicit assignment of non-deterministic values to the input signals. The software-netlist model retains the module hierarchy of Verilog RTL. Thus, each clock step in hardware is simulated by a call to the top-level function in the software-netlist. Due to space limitations, we only briefly explain the translation here and refer the reader to *v2c* for more details.[3]

B. Cycle-accurate and bit-precise translation

v2c translates Verilog RTL to a cycle-accurate and bit-precise model in C. Due to the sequential nature of the software-netlist model, the tool performs intra-modular and inter-modular dependency analysis between the clocked blocks containing procedural statements (blocking or non-blocking) and the continuous assignment statements. The generated C code conforms to the synthesis semantics of Verilog RTL.

[1]http://fmv.jku.at/hwmcc14cav/

[2]http://sv-comp.sosy-lab.org/2015/index.php

[3]http://www.cprover.org/hardware/v2c

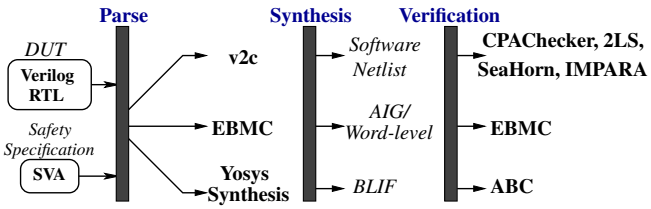

Fig. 2. Tool flow for hardware property verification

Verilog has few operators—such as part-select, bit-select from vectors, concatenation, reduction—that are not available natively in C. *v2c* translates these to semantically-equivalent C expressions using an appropriate combination of C operators.

It is straightforward to generate a software-netlist model for circuits without feedback loops. However, for designs with loops, there are two possible ways to synthesize a word-level software-netlist model. The first approach is to create a flattened software-netlist model by decomposing the module hierarchy, in a way similar to how a word-level transition system would be synthesized. An alternative is to retain the modular structure by analyzing the conditions under which signal values in combinatorial logic reach fixed-points. We aim to take the second approach, since it retains the word-level structure as well as the module hierarchy of the Verilog RTL. Currently, however, *v2c* does not handle combinational loops, transparent latches and designs with multiple clocks.

C. Equivalence of Verilog and Software-netlist

In order to obtain a trustworthy translation, we refrain from doing synthesis-based optimization and abstraction during the translation of Verilog RTL to a software-netlist. While we do not have formal evidence that our translator is correct, we have not observed any mistranslations during our experiments with various control and data-dominated benchmarks. On the unsafe benchmarks, the bug is manifested in the same clock cycle for both models; on the safe benchmarks, the properties are proven to be k-inductive with the same value of k for both models.

D. Software Verifiers at the heart of Hardware Verification

Figure 2 shows the tool flow for performing hardware verification at bit-level, word-level, and software-level. We use ABC to perform bit-level unbounded safety verification. ABC does not support Verilog, so we use an open-source synthesis tool, YOSYS 0.5[4] to translate Verilog RTL to BLIF, which is then passed to ABC for verification. For word-level unbounded verification, we use word-level k-induction engine of EBMC, which supports IEEE 1364.1 Verilog 2005. For the software-netlist verification flow, we use our tool *v2c* to generate a software-netlist model from Verilog. We then apply a wide range of representative unbounded software verification techniques to determine the safety of these benchmarks. In particular, we use k-induction [21] (implemented in the tools CBMC [11] and 2LS [5]), interpolation (CPAChecker [1], IMPARA [23] implementing the IMPACT algorithm [19]), abstract interpretation (Astfee [3]) and IC3/PDR (SeaHorn [13]).

[4]http://www.clifford.at/yosys/

IV. Experimental Results

In this section, we report experimental results for unbounded safety verification of hardware circuits at various levels of design granularity, as shown in Figure 2. We compare state-of-the-art hardware model checking tools, such as *ABC 1.01* (winner of HWMCC 2014 in safety track) and EBMC 4.2, with various software analyzers from SV-COMP 2015, such as *CPAChecker 1.4*, *SeaHorn (revision 07666c810d)*, *2LS 0.3.4* and a classical abstract interpretation based tool like *Astrée*.

Our experiments were performed on an Intel Xeon machine running at 3.07 GHz. We restricted the resources to 5 hours and 32 GB RAM per benchmark. To enable other researchers to reproduce our results, all our benchmarks in Verilog, software-netlist models in ANSI-C, scripts for running YOSYS 0.5, ABC, EBMC 4.2, and other software verification tools are uploaded to a publicly accessible archive website.[5]

Benchmarks: We target two different classes of circuits: data-path intensive circuits, including a Huffman encoder/decoder and a Digital Audio Input-Output chip (DAIO); and control-intensive designs, including a non-pipelined 3-stage processor, a Read-Copy-update mutual exclusion protocol, a FIFO controller, a buffer allocation model, and an instruction queue controller. The safety properties are specified as System Verilog assertions (SVA). The properties are instrumented as assertions in the software-netlist model. The benchmarks in our paper are derived from real world hardware benchmark suites, including VIS Verilog models, the Texas-97 Benchmark suite, and opencores.org.

Discussion: Figures 3–5 report the comparison of various unbounded verification techniques employed by verification tools at bit-level, word-level, and software-level. We categorize the approaches into three classes:

- *k*-induction (Figure 3)
- interpolation (Figure 4), and
- PDR together with other hybrid techniques (Figure 5).

By hybrid techniques, we refer to predicate abstraction as implemented in *CPAChecker* and a combination of *k*-induction, BMC and abstract interpretation as implemented in *2LS* [5]. On the *x*-axis is the analysis time in seconds and on the *y*-axis we list the benchmarks. The vertical red lines on the right-hand side of the diagrams show timeouts, out of memory, inconclusive (unknown) results, errors (crashes), and wrong results (tool bugs) reported by the tools. The tools can be distinguished by the size of the circles as well as by colour.

Analysis using k-induction: For safe benchmarks, the results for bit-level, word-level verifiers and software verifiers are comparable when the properties are 1-inductive or 2-step inductive. However, for complex safety properties, ABC and other abstraction based software analyzers either timeout or took a long time to terminate. We investigated the reason for higher verification times for some safe benchmarks, such as the FIFO controller, the RCU, and Buffer Allocation. We observe that the properties are not *k*-inductive for sufficiently large values of *k*, e.g. (k=1000) and thus tools based on *k*-induction either timeout or took long time to compute the inductive invariant sufficient to prove the property. For the unsafe benchmarks, for example DAIO and the traffic light controller, where the bugs are manifested only at 64 and 65 clock cycles respectively, the verification times using ABC and EBMC's *k*-induction engine are comparable to CBMC 5.2 and 2LS. Figure 3 reports the time taken by the *k*-induction engine in ABC, EBMC 4.2, CBMC 5.2 and 2LS. We did not report the time for *CPAChecker* since the results suggest that its *k*-induction engine is not as mature yet.

Analysis using Interpolation: Figure 4 reports the time taken by the interpolation engine in ABC, IMPARA and *CPAChecker*. ABC is the fastest in 9 out of 12 designs. However, it times out on three complex benchmarks, RCU, FIFO and BufAl, whereas the software interpolation tool, IMPARA, which implements IMPACT algorithm solved three instances out of which one is the complex FIFO design; yet IMPARA either timed out or ran out of memory for the remaining designs. *CPAChecker* solved 5 out of 12 cases. None of the interpolation engines was able to prove RCU and BufAl.

Analysis using Hybrid techniques: Figure 5 reports the time taken by the IC3/PDR engine in ABC, *SeaHorn* and other hybrid techniques as implemented in *CPAChecker* and 2LS. ABC is the clear winner here; it is the only tool that proves the FIFO and BufAl benchmarks safe within the given 5h timeout. *SeaHorn*'s PDR engine solves half of the benchmarks, but produces false negatives on the other half due to limited support for bitvectors. 2LS successfully solved 8 benchmarks and times out on four benchmarks. *CPAChecker*'s predicate abstraction reliably solves 7 benchmarks, but times out on two benchmarks and reports three wrong results. Note that none of the tools was able to prove RCU. We do not report the results using Astrée since it requires manual directives for data and control partitioning to avoid imprecision; nonetheless it generates many false alarms for safe benchmarks.

Summary of the results: We investigated the reason for large number of timeouts, wrong results and errors produced by the software verifiers. We observed that software-netlists heavily use bit-level operations and thus bit-precise reasoning ability is necessary for the underlying verification engine. However, bit-level operations are less prevalent in conventional software and hence less tested in software analysis tools. Also, software verification tools often use numerical abstractions, which are likely to lose important bit-precise information. As a consequence, our results show that running conventional sotware verification tools on software-netlists exhibits many tool bugs ("wrong").

The abstraction and invariant inference techniques employed in software tools such as *CPAChecker* and *2LS* have never been optimized for hardware analysis. But the results in this paper show that these tools are within one order of magnitude compared to hardware model checkers for detecting bugs or proving safety for some of the software netlist models. We thus believe that there is scope here for new tools that implement abstract interpretation using abstract domains developed specifically for this task, e.g. by applying abstract conflict driven learning [12].

V. Conclusions

We present an approach to unbounded safety verification for hardware designs given in Verilog RTL, at the heart

[5] http://www.cprover.org/hardware/date2016/

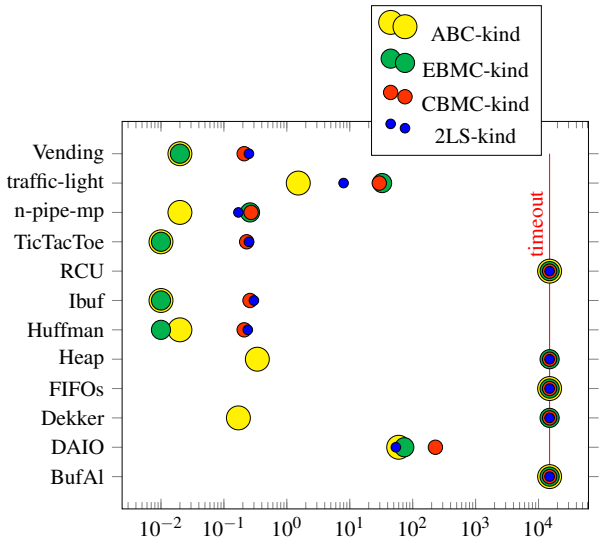

Fig. 3. Comparison of k-induction tools

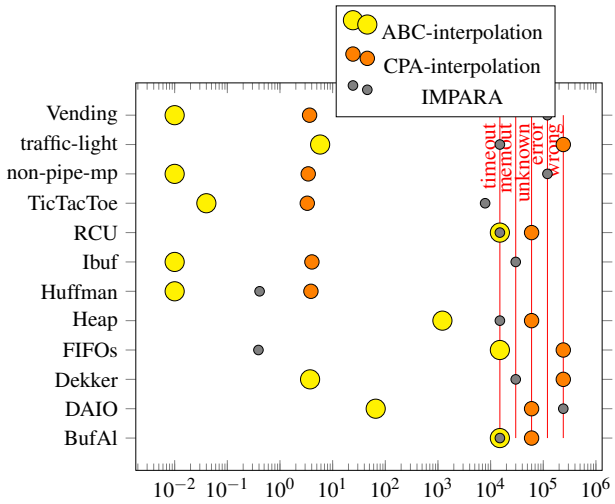

Fig. 4. Comparison of interpolation-based tools

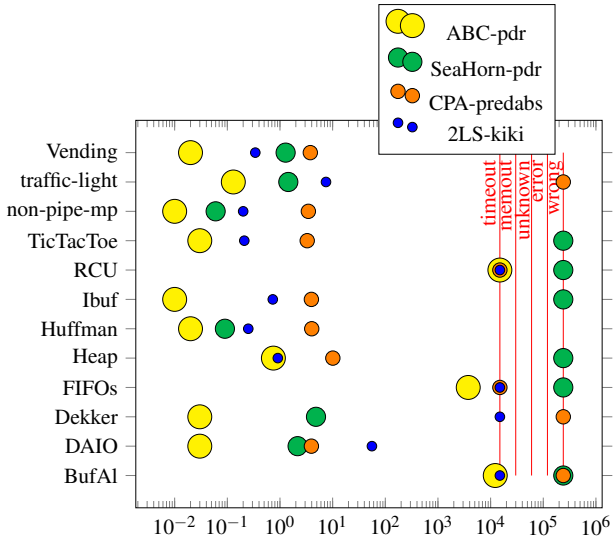

Fig. 5. Comparison of hybrid techniques

of which are software verifiers. We present a comparison of unbounded verification techniques at bit-level, word-level and *software-netlist* level. The range of software verification techniques is vast; this paper can thus only be an initial step. We will evaluate further software verification techniques and their application to hardware property checking and co-verification workloads as future work.

References

[1] Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: CAV. LNCS, vol. 6806, pp. 184–190 (2011)

[2] Bjesse, P.: A practical approach to word level model checking of industrial netlists. In: CAV. LNCS, vol. 5123, pp. 446–458 (2008)

[3] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI. pp. 196–207. ACM (2003)

[4] Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: FMCAD. pp. 173–180 (2007)

[5] Brain, M., Joshi, S., Kroening, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: SAS. LNCS, vol. 9291, pp. 145–161. Springer (2015)

[6] Brayton, R.K., Hachtel, G.D., A.Vincentelli, Somenzi, F., Aziz, A., Cheng, S., Edwards, S.A., Khatri, S.P., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R., Sarwary, S., Shiple, T.R., Swamy, G., Villa, T.: VIS: A system for verification and synthesis. In: CAV. pp. 428–432 (1996)

[7] Brayton, R.K., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: CAV. LNCS, vol. 6174, pp. 24–40. Springer (2010)

[8] Chakraborty, S., Khasidashvili, Z., Seger, C.H., Gajavelly, R., Haldankar, T., Chhatani, D., Mistry, R.: Word-level symbolic trajectory evaluation. In: CAV. LNCS, vol. 9207, pp. 128–143. Springer (2015)

[9] Chatterjee, S., Kishinevsky, M.: Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. Formal Methods in System Design 40(2), 147–169 (2012)

[10] Cimatti, A., Griggio, A.: Software model checking via IC3. In: CAV. LNCS, vol. 7358, pp. 277–293. Springer (2012)

[11] Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988, pp. 168–176. Springer (2004)

[12] D'Silva, V., Haller, L., Kroening, D.: Abstract conflict driven learning. In: POPL. pp. 143–154. ACM (2013)

[13] Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: CAV. vol. 9206, pp. 343–361 (2015)

[14] Jain, H., Kroening, D., Sharygina, N., Clarke, E.M.: Word level predicate abstraction and refinement for verifying RTL verilog. In: DAC. pp. 445–450 (2005)

[15] Keating, M.: The Simple Art of SoC Design. Springer (2011)

[16] Kroening, D., Seshia, S.A.: Formal verification at higher levels of abstraction. In: ICCAD. pp. 572–578 (2007)

[17] Kroening, D., Weissenbacher, G.: Lifting propositional interpolants to the word-level. In: FMCAD. pp. 85–89 (2007)

[18] Liu, L., Vasudevan, S.: Scaling input stimulus generation through hybrid static and dynamic analysis of RTL. TODAES 20(1), 4:1–4:33 (2014)

[19] McMillan, K.L.: Interpolation and SAT-based model checking. In: CAV. LNCS, vol. 3114, pp. 1–13. Springer (2003)

[20] Mukherjee, R., Kroening, D., Melham, T.: Hardware verification using software analyzers. In: ISVLSI. IEEE (2015)

[21] Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a sat-solver. In: FMCAD. vol. 1954, pp. 108–125 (2000)

[22] Sülflow, A., Kühne, U., Fey, G., Große, D., Drechsler, R.: Wolfram-A word level framework for formal verification. In: RSP. pp. 11–17 (2009)

[23] Wachter, B., Kroening, D., Ouaknine, J.: Verifying multi-threaded software with impact. In: FMCAD. pp. 210–217 (2013)

[24] Welp, T., Kuehlmann, A.: Property directed invariant refinement for program verification. In: DATE. pp. 1–6 (2014)